**Politico – Final Year Project**

Building a Cross Platform, Component Driven Game for the Modern Web

Joshua Jackson (P16179167)

Supervisor: Dr. Archie Singh Khuman (Arjab.Khuman@dmu.ac.uk)

Acknowledgements

*[…See example material for how to best structure this.. archie for supervision, and any resources that I made heavy use of (not just one offs)]*

Table of Contents

*[..To be generated using headings]*

Introduction

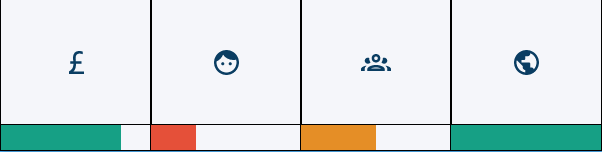
What is the game?

The project is a browser-based cross platform strategy game where the goal is, as the leader of the United Kingdom, to keep the attributes of the nation (financial, population happiness, domestic and foreign political favour) balanced for the full length of a premiership. The way in which a user achieves this goal is by making yes/no decisions that alter the nation’s attributes in different ways where the user must weigh up the cost of a decision in both the long term and the short term.

The decision-making system is the only mechanic for interacting with the game that the user has which results in simple gameplay that becomes increasingly more complex the more strategy is applied by the user.

Attributes

All the nation’s attributes are adjusted whenever the user makes a decision. These can be seen in [Figure 1].



[Figure 1: The nation’s attributes as shown in the game’s UI]

Financial

Decisions can have a *Very Positive, Positive, Neutral, Negative* or *Very Negative* financial impact. This will be applied to the overall financial attribute when a decision is made. If a decision, for example, has a negative impact and the user decides *not* to go through with it (by selecting No), there will be no financial impact whatsoever.

Population Happiness

Each decision has a political leaning associated with it which is checked by the political leaning of each region in the nation, the more deviation a certain decision has from a region’s political leaning, the more happiness is lost and, conversely, the closer the political leaning of a decision to the leaning of the region, the more happiness is gained. The user can gauge the political leaning of the nation by clicking through the regions on the map, each region will have a rough estimation of its political leaning, the user will also be shown the various factors of the region that determine its political leaning such as population density, number of universities etc (see: Fuzzy Logic section for how this is calculated).



[Figure 2: Screenshot of the region information modal]

Domestic Political Favour

Domestic political favour works the same way as population happiness, however it is only limited to the regions that are represented by a member of the user’s party. The regions that are represented are selected at random when the game first starts. To keep this attribute balanced, the user must consider both the overall nation’s political leaning but also the specific leaning of the regions represented by their party. As seen in [Figure 3], the regions represented by the user’s party are highlighted in green.

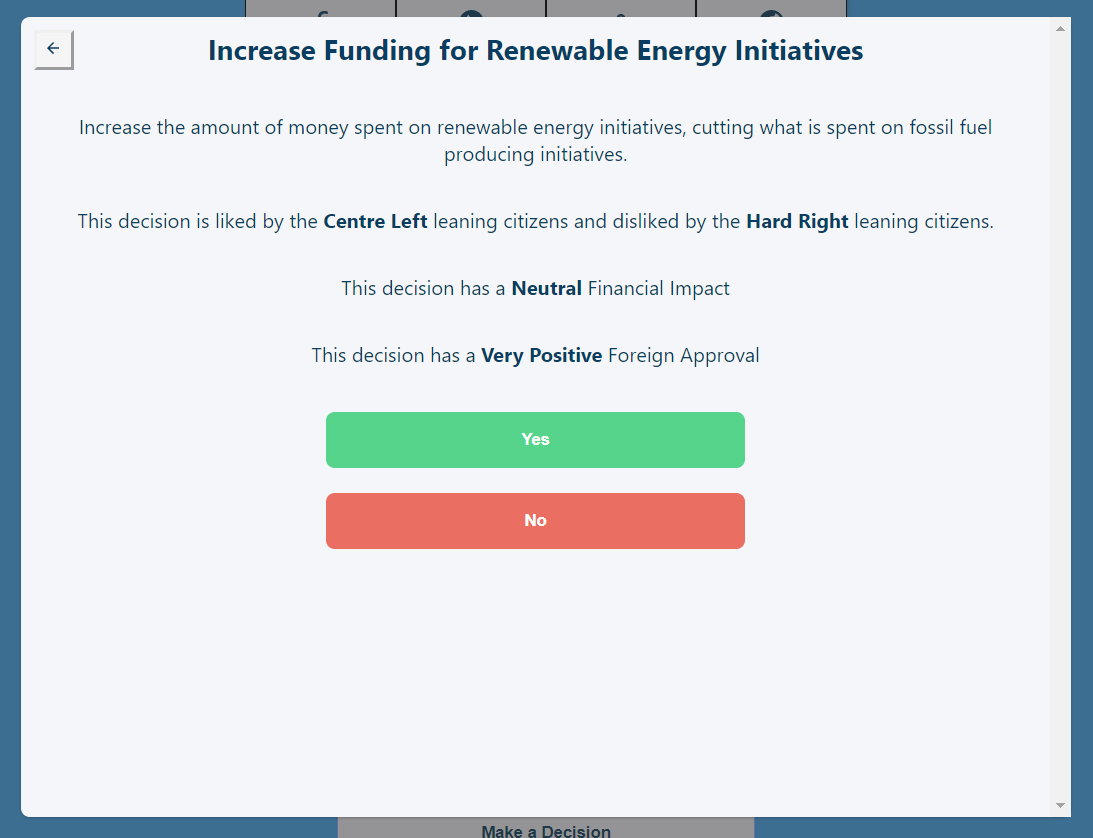
A picture containing text, map

Description automatically generated

[Figure 3: Screenshot showing the domestic political favour attribute and regions represented by the user’s party]

Foreign Political Favour

The foreign political favour works much like the financial attribute where a decision lists the impact of a certain decision as Very Negative to Very Positive with regards to how the rest of the world sees the decision. For example, a decision enacting a policy of the European Union may have a very positive impact on foreign political favour but might have a negative financial impact, these are all options the user must weigh up before deciding.



[Figure 4: Screenshot of a decision being made]

Inspiration

The main inspiration for this project is the game Reigns **[1]** which uses a similar decision-making system as the sole mechanic. The similarities, however, are quite different beyond the core mechanics.

The main differentiator between this project and its competitors is the cross-platform functionality (ability to use across all devices) as well as the gameplay nuances powered by a research-backed fuzzy logic implementation.

Why was this project undertaken?

Work was undertaken on this project mainly to develop my own skill within the areas of frontend web development, specifically around the test-driven development methodology.

As the game is entirely based around the user interface, it provided the perfect basis for a highly testable project and because it was created using the latest modern web technologies it also allowed the use of the most modern test tooling to give me the best chance to learn best practices before going into the industry.

Another reason for this project is the goal of building a fully-fledged, albeit simple, strategy game that is accessible from anywhere and on any device*.* This is an area which is notoriously lacking within strategy games. The challenge in making a game that strikes a good middle ground and that can run anywhere without changing any of the gameplay was a main motivator.

Finally, the ability to incorporate current affairs that a user may have heard or read about into a game where the user is the one actually making the decisions was a big factor in the conception of this project.

Analysis of Requirements

Project Problem Domain

A clear explanation of the problem …

This project sits between strategy games that provide an enjoyable experience but are limited in their accessibility and games that are easily accessible but lack enjoyable gameplay to keep players interested.

Many strategy games have intriguing gameplay but have an incredibly high learning curve where the enjoyability of the game is hidden behind a multitude of features and tools that the user must have experience with in order to properly play the game. Furthermore, these games are usually only playable on a desktop device leading to an entire market of potential users not having the ability to experience the game. An example of such a game is Hearts of Iron **[2]** as seen in [Figure 1].



[Figure 5: Screenshot of Hearts of Iron]

On the flip side, many strategy games are very accessible by providing simple gameplay and the ability to play across devices (desktop, tablet and mobile) but these games often start out by relying on gameplay features for their enjoyability which they then remove to make the game simple to play, resulting in a game that is easy to use, but lacks enjoyment for the end user. Many successful games such as Civilization **[3]** manage to strike a balance between ease of use and enjoyable gameplay and even offer cross platform ports of the game but, even though that is the case, they fail to provide an equivalent experience across devices, often favouring desktop users with the mobile/tablet ports being an afterthought.

Given the landscape, this project aims to provide an easy to use experience, coupled with enjoyable gameplay that is developed and tested across all relevant platforms simultaneously to provide the user with comparable cross platform gameplay whilst also relying less on features and more on core gameplay.

Core Aims and Objectives

A clear explanation of the … objectives

Core Aims

The core aims of this development project are to deliver a cross platform, easy to use game that works on modern desktop, tablet, and mobile devices all from a single codebase as well as to explore the practical use of fuzzy logic within video games with the goal of providing an experience that has more depth and nuance.

Gameplay Aims

The main aim of the gameplay is to provide the user with an enjoyable experience all based around their approach to linking various non-directly related pieces of information together to help make an informed decision without having many features getting in the way, allowing the user to essentially “create their own gameplay” much like games such as EVE **[4]** or Supremacy 1914 **[5]** but on a smaller scale where gameplay is driven by the players themselves.

The gameplay also aims to draw contrasts to the real-world with the decision-making process where the user may have heard about the decisions they are making on the news or internet further adding to the enjoyability of the game.

Finally, the gameplay is designed to be enjoyable in small bursts, where the user may wish to only play a few turns (make a few decisions) and then put the game down or during longer sessions with the user playing multiple full games.

Development Aims

The project also aims to make extensive use of modern web technologies such as React **[6]** and TypeScript **[7]** alongside testing tools such as Cypress **[8]** and Enzyme **[9]** as well as comprehensive user testing resulting in an enjoyable gameplay experience and a stable, maintainable, and future proof codebase consisting of many reusable UI components.

Major Components of the Game

 A clear explanation of each of the major aspects of the product, what the requirements/aim were and were they met.

Cross Platform/Responsive Design

The game is designed to function on desktop, tablet and mobile without hiding or degrading any of the functionality from the user. This is mostly achieved by implementing a responsive gameplay viewport that dynamically scales and positions elements relative to the screen size of the device.

The requirements for the game were that it could be opened on a large desktop monitor and then the window could be scaled down slowly without breaking at any point in the process. The minimum screen size that is supported is 320x568 as there aren’t any devices still in use that are smaller than this. As well as a correctly scaled user interface, the game should also perform well on any device and not provide the user with a slower experience because their device is older.

These requirements have been met mostly using well-structured HTML and responsive CSS and are further solidified in place using end to end testing (see: Test Driven Development) where the resizing of the user interface and the speed of the application is tested before any code can be pushed to the remote repository, if a test fails, the code that has broken it must be fixed.

Fuzzy Logic

Fuzzy logic is one of the main components as the game deals with a lot of “guess-work” with regards to determining where on the political spectrum both decisions and the views of the population in the game lie. As a basis, this gameplay element was implemented randomly but, after extensive research into how fuzzy logic may be “used within less serious situations such as within Video Games that feature the simulation of political leaning” **[10]**, the main focus was on attempting to implement fuzzy logic in a practical sense within the domain of political leaning in video games.

The requirements were as follows:

* Each region within the game (on a map of the UK) should be given various factors that would determine their political leaning. These factors are: population density, number of universities, non white-British ethnic percentage and average salary.
* Whenever a decision is made in the game, each region’s factors would be passed into a fuzzy inference system that will return a defuzzified political leaning output.
* The leaning of each region would then have an impact based on how the decision is answered, where regions that lean closer to the decision on the spectrum will gain happiness and those who lean further away will lose happiness.

Fuzzy logic has been fully implemented into the game using C# and the Fuzzy Logic Sharp library **[11]** and has produced accurate results\*, albeit only after various stages of tuning as can be seen in the testing section of the appendix.

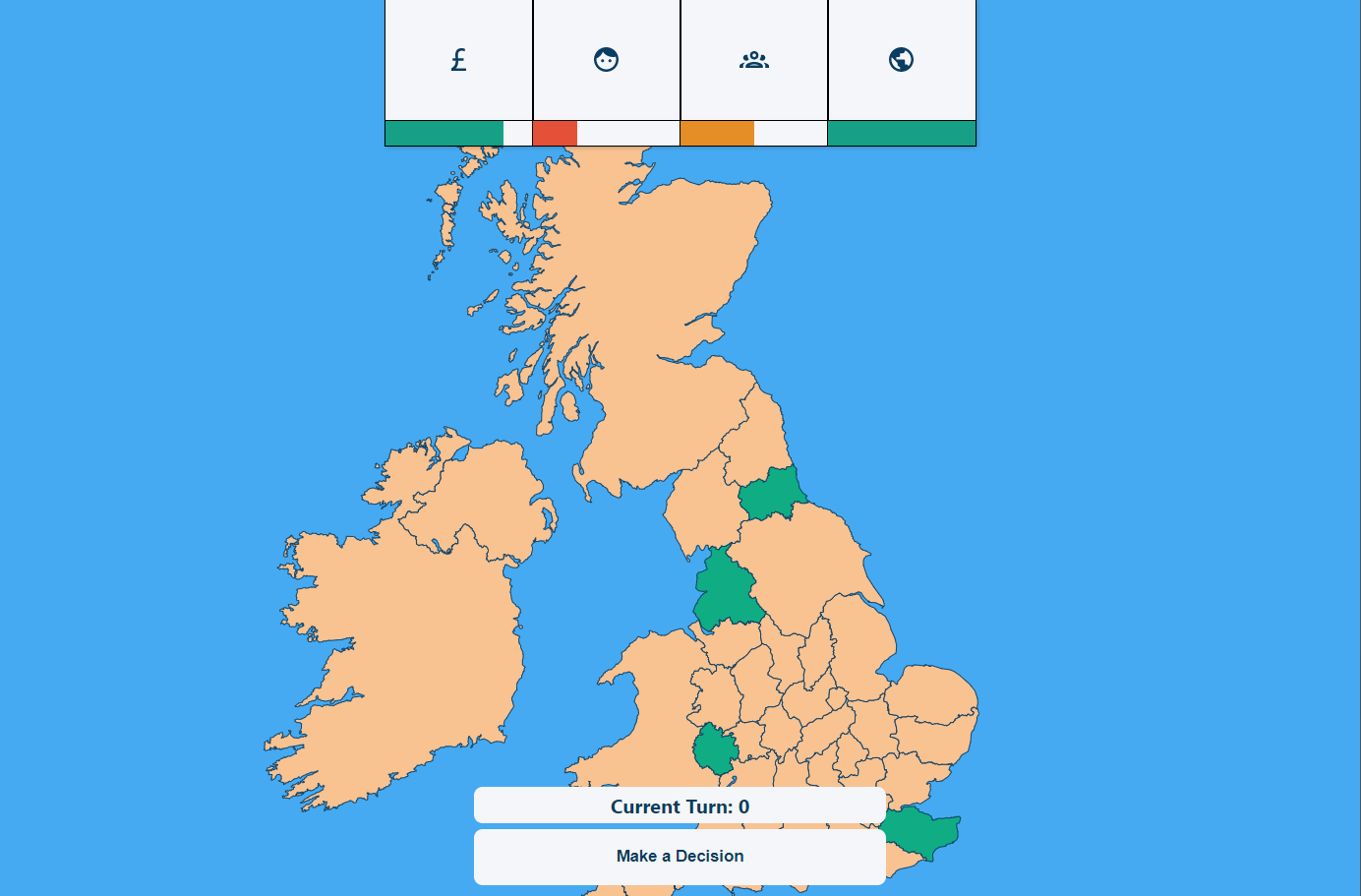
\*Accurate based on conclusions reached in research carried out by Joshua Jackson for the IMAT3406 Fuzzy Logic & Knowledge Based Systems module.

User Interface and Controls

The user interface (UI) is the only way through which the user interacts with the game as there are no keyboard or gesture controls beyond being able to zoom in and pan around the map. There aren’t many interactions that the user can carry out within the game but those that exist are as follows:

* The user can click on a region within the game to be shown more information about it.
* The user can click on the Next Turn/Make a Decision button to be shown the decision modal.
* The user can click Yes or No on the decision modal to make a decision.
* The user can use the mouse wheel or pinch the touch screen to zoom in/out on the map.
* The user can click/touch and drag to pan around the map.

The user interface meets all the functional requirements outlined above, and it is also designed to be clean and non-intrusive as one of the main aims of the game, as outlined initially, is to not get in the way of the user. A brief screenshot of the user interface can be seen in [Figure 2], full screenshots can be found in the appendix.



[Figure 6: Screenshot of the user interface in the project]

Major Components of the Development

A clear explanation of each of the major aspects of the development process overall, what the requirements/aim were and were they met.

Test Driven Development

The development process adopted for this project is Test Driven Development (TDD) where every feature has automated tests written around it as a first-class citizen. The goal of this methodology is to ensure that the code being written at any given moment cannot be easily broken by code written in the future as there will be automated tests in place that check the functionality works as expected, these tests are ran before any new code can be committed to the remote repository, ensuring a broken version is never deployed. Another major benefit of this approach is much more maintainable and readable code; this is because, for a component to be easily testable, it must be written in a verbose way which also has the effect of making code easier to read and reason about. The tests also provide peace of mind when coming to make changes to existing code as the test will inform the developer if anything has been broken as a result of their changes.

Implementing TDD in this project has proven to provide the benefits outlined above, the largest being the safety to refactor code as there have been multiple relatively large refactors of existing code during the project development lifecycle made with the confidence that TDD provides where these improvements may not have been made had the safety of automated tests not been present due to the fear of breaking vital parts of the application.

Modern Web Technology

One of the pillars of this project is the use of modern web technology as the basis for the entire application. This meant using technologies such as React **[6]** and TypeScript **[7]** which came with many benefits, the most prominent being the developer experience, using new technologies brings with it all of the latest development patterns which often results in a better experience as these technologies have been created with the modern developer in mind, leading to a more enjoyable experience, which leads to a better final product.

The use of modern web technology has been an overall pleasant experience, only causing slight issues due to the lack of resources online for edge cases as the technology is so new.

Project Management

Clear evidence of the application of an appropriate software development methodology. Discussion of each of the major stages including how validation and verification were applied at each stage.

A significant part of this project was the project management and how this was implemented, including the development lifecycle chosen and the software or tools used, played a huge role in the success of the project.

Development Lifecycle

This project made use of an ***agile*** sprint-based methodology wherein the development of the project is first split into Epics which are based on each month of development (eg. January Epic). These Epics are then further split up into sprints, usually containing 2-3 sprints around a week and a half long.

At the end of each sprint, a sprint retrospective is carried out which covers what went well, what could have been improve and any actions to be carried out as a result. If an action is easily resolved, it would be added to the upcoming sprint.

At the end of each epic, there is an epic retrospective which follows a similar process to a sprint retrospective but on a much higher level and usually whilst thinking about the overall direction of the project rather than specific tasks. Each epic retrospective features a backlog refinement wherein tasks are taken from an existing list of backlog items and put into the relevant sprints for the next epic.

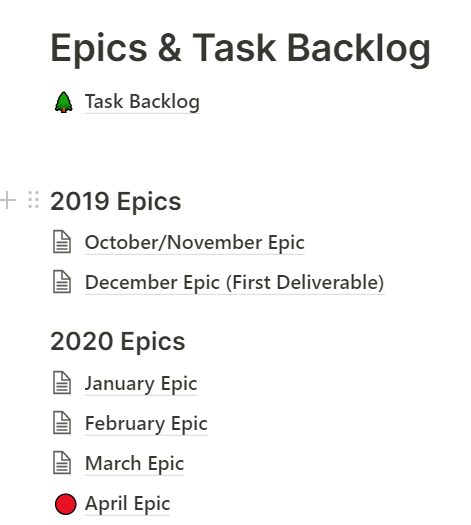
The task backlog is a vital piece of the development lifecycle, if a task cannot be completed within a given sprint, it will be added to the task backlog to then be picked up when organizing future sprints and epics.

To gain a form of deliberate practice **[12]** at the end of each sprint, the retrospective would be sent to the project supervisor to keep a log of the progress being made on the project and allow for feedback with regards to the direction the project is heading. In the same vein, at the end of each Epic, where possible, user testing sessions would be set up to gain vital feedback on the gameplay as well as any obvious defects in the functionality (see: Testing section)

Software and Tools

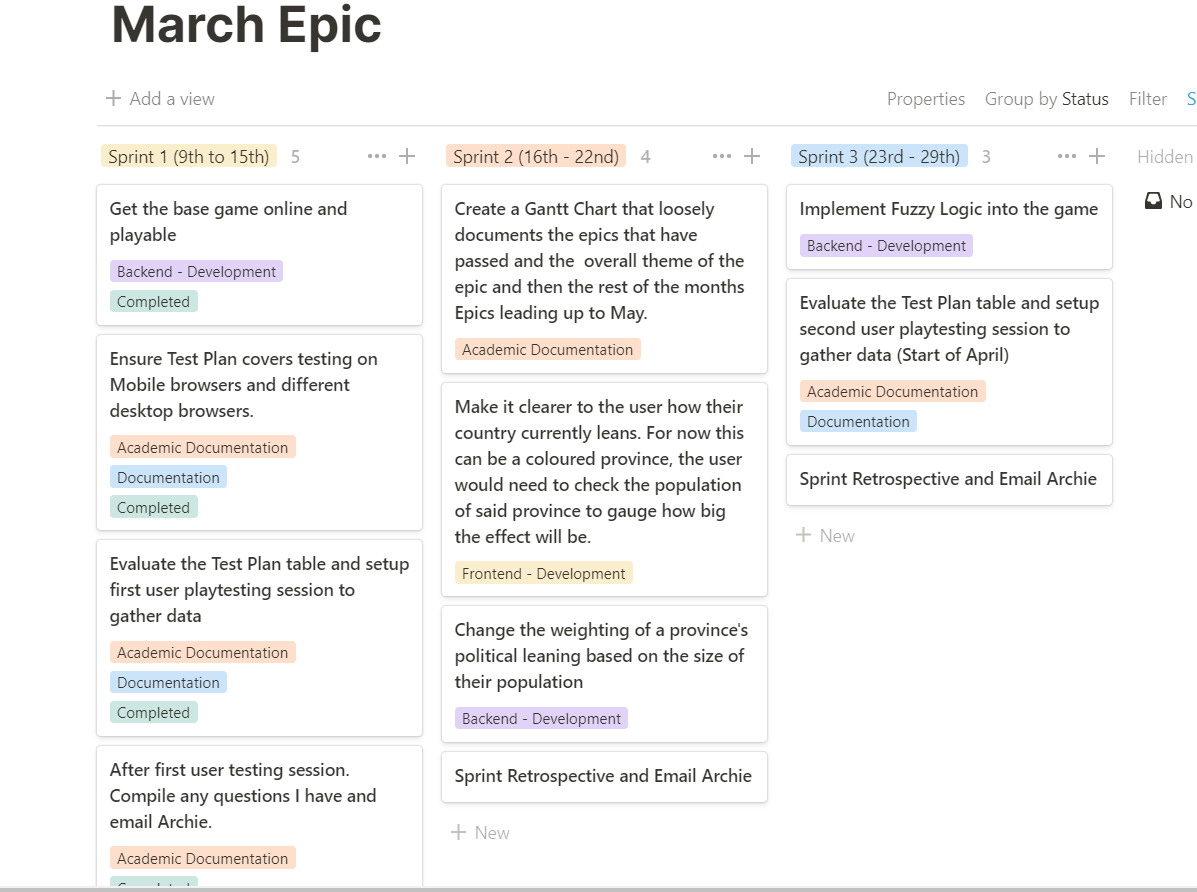
To facilitate the agile development lifecycle outlined above, proper software and tooling needed to be used as the process would only be as good as the systems that were put in place around it.

The first, and most prominent, piece of software was Notion **[13]**; This tool acted as an all-in-one wiki and Kanban board which allowed the grouping of relevant information together. For example, in [Figure 7] the task backlog page is situated right next to the epics for each month.



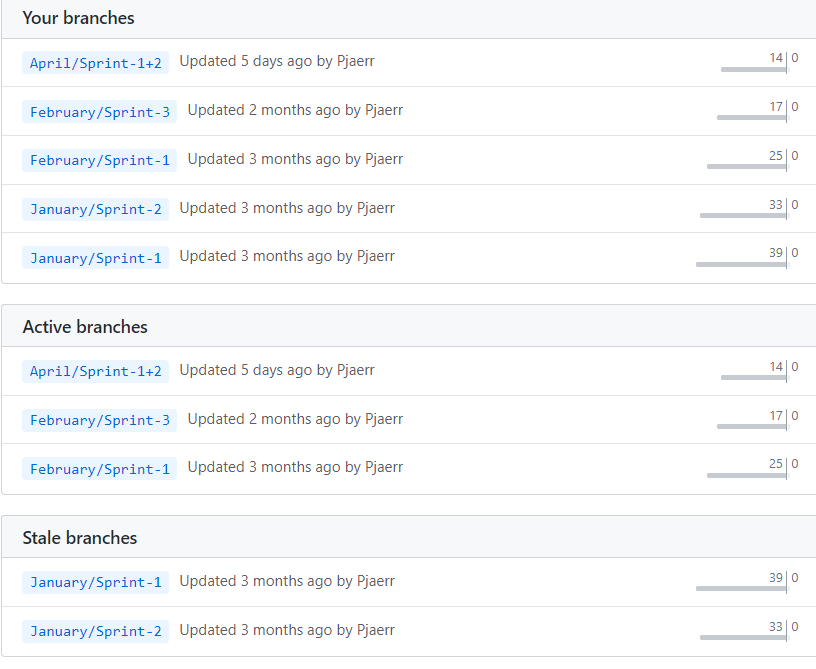
[Figure 7: Screenshot of the Notion homepage]

Notion also featured an easy to use Kanban board which allowed the tagging of tasks so that the major properties of each piece of work could be seen at a glance, the Kanban board was used for each epic, where each column is a sprint within that epic. This can be seen in [Figure 8].



[Figure 8: Screenshot of the March epic inside of a Notion Kanban board]

Github **[14]** was used for the entire duration of project for managing the code as well as for breaking down large pieces of work into manageable chunks that could be easily linked with relevant tasks on the Kanban board[s]. This was achieved mainly through the use of a regimented branching structure where, for each sprint within an epic, a branch would be created with the following structure: MONTH-Epic/Sprint-NUMBER and then, at the end of that sprint, the branch would be merged into the main development branch. This not only provides structure, but also makes it easy to revert to a specific version of the codebase for each respective sprint. An example of the branching structure can be seen in [Figure 9].



[Figure 9: Screenshot of the branching structure used on Github]

Design

rationale for each of the design … decisions.

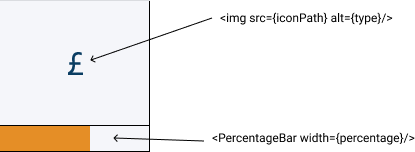
Frontend Application Architecture

User Interface

Component Driven Design

The architectural approach taken for the frontend revolves heavily around user interface components where each piece is split into composable components. These components can be placed anywhere on the webpage and take properties that can then be used to define change in the component from outside of the component.

An example of this would be the <Attribute/> component that takes the following properties: {type: string, iconPath: string, percentage: number}. This component is designed to be reusable and that is exactly how it is used. For each attribute, an instance of the <Attribute/> component is put onto webpage, except each component instance will have a different value passed in as its properties.

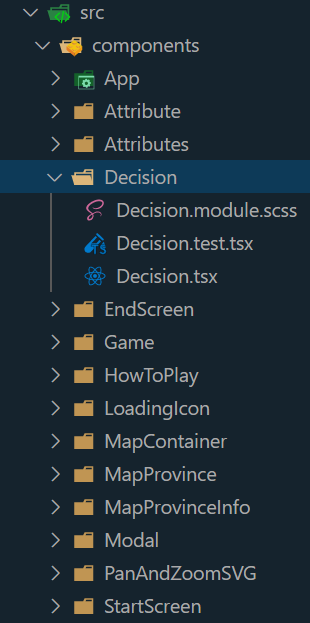


[Figure 10: The <Attribute/> component]

This structure has the main benefit of making the code *DRY* **[15]** meaning that code isn’t repeated which also means there is a single source of truth. Sticking with the example above, if a change needed to be made to the design of the attributes, it would only need to be made in a single place and would be applied to all instances of the component.

The components are also composable, meaning components can have components inside of them, this can be seen in [Figure 10] where the <Attribute/> component uses a <PercentageBar/> component, further increasing the reusability.

Having a component-driven architecture means that all relevant code for a component is stored together, this includes the HTML, JavaScript and CSS. This not only makes maintaining the codebase easier as the developer doesn’t need to search through the entire codebase to find where code is being used, it also makes working on and testing components in isolation incredibly easy. An example of this can be seen in [Figure 11].



[Figure 11: A screenshot of the project’s folder structure with components in their own folder, encapsulated with the HTML, CSS and JavaScript they need to function as well as any tests]

Shortcomings

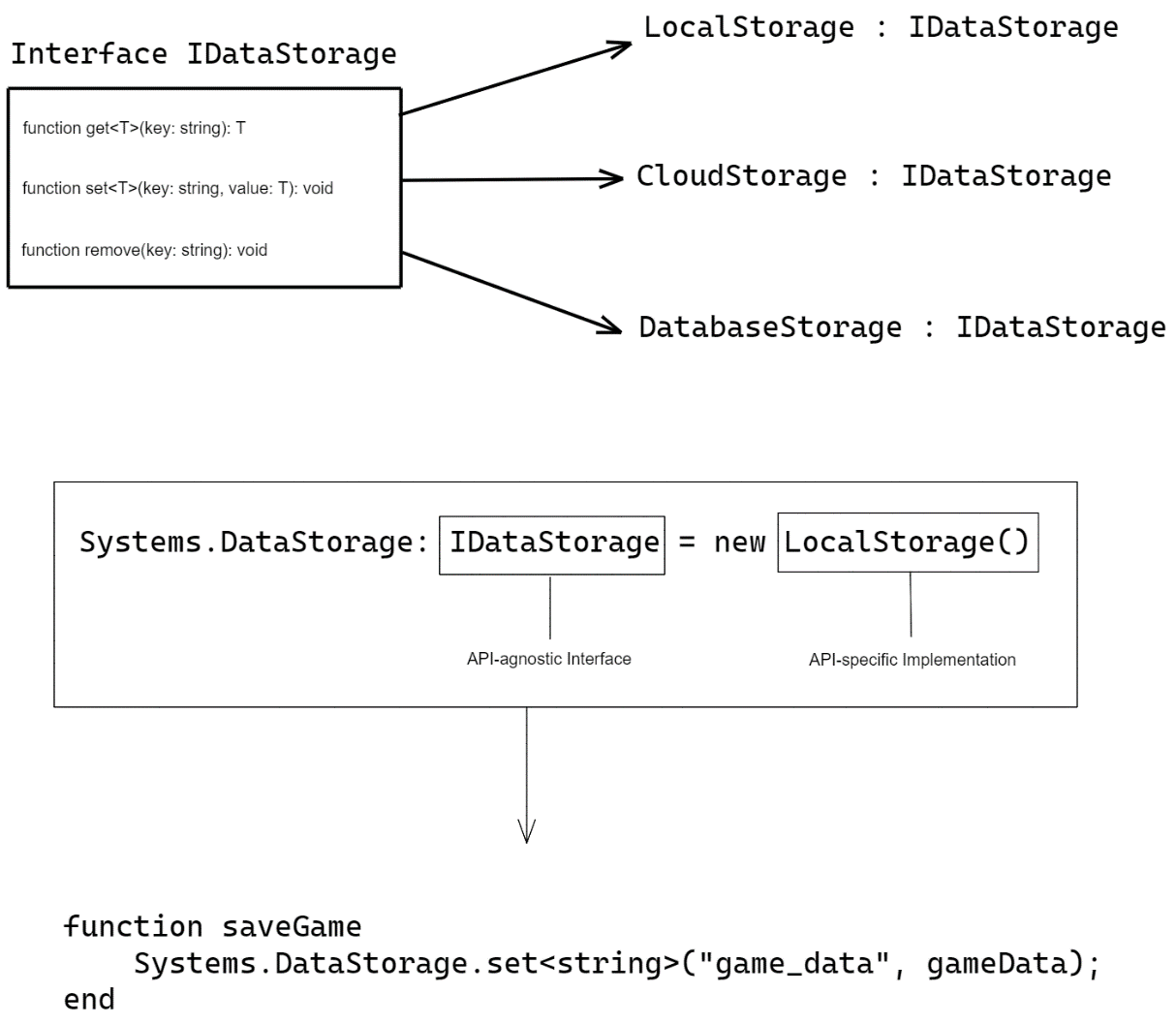
There aren’t many issues with the component driven approach, and it is one the most used architectures on the modern web for a good reason. Even though that is the case, it still has some shortcomings. As everything must be isolated, it can often take longer to implement parts of a user interface than it normally would as the developer must ensure that nothing leaks into other parts of the codebase.

Another downside is that often multiple components share a component, an example of this is a Modal component that many components may share because the implementation details are quite long and would be non-DRY if repeated; if one of the components wanted to slightly tweak the Modal but only for itself, it wouldn’t be able to easily without either passing in styling choices as properties which can quickly pollute the component, or by breaking out of the Modal component, thus rendering the point of the component driven architecture useless.

Logic

API-Agnostic Systems

The UI is driven by components but the logic on the frontend is instead driven by API-agnostic systems. This means the use of a generic TypeScript interface **[16]** for each system and then the places where the system is used are not concerned about the implementation details of the underlying system, they only know about the functionality the interface exposes, allowing the underlying system to be completely swapped out for something different without having to change how the code works, as long as the underlying system matches the interface. This structure can be seen in the diagram in [Figure 12] which demonstrates the DataStorage system.



[Figure 12: Diagram showing how the IDataStorage interface works at a high level]

As can be seen in [Figure 12], new LocalStorage() could be swapped out for new CloudStorage() and nothing would need to change, just the implementation under the hood.

Shortcomings

The architecture described above is extremely powerful and very easy to reason about however the main shortcoming is that there is a high likelihood that the system interfaces may become too rigid and it will become harder to add changes to an interface.

Using the example of the LocalStorage and CloudStorage implementations. LocalStorage only supports very few functions but CloudStorage supports many more due to the nature of how it works, this may become hard to express both systems within a single interface.

Backend Application Architecture

* How the backend (C#) is structured from an architectural POV and \_why\_
* Serving static assets (the frontend) instead of separate routes.
* Mention any new concepts that have been learnt/solidified as a result of researching possible architectures.

User Interface Design

* Tools used
* Thought process and inspiration
* Mention any new concepts that been learnt/solidified as a result of research.

Implementation

rationale for each of the … implementation decisions.

Overview TSDOC diagram here (UML plant whatever..)

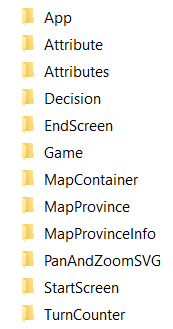
Core Gameplay Mechanics

Talk about the core gameplay mechanics/functionality, not mentioning specific components (listed below).

Functionality and User Interface

(what I refer to as the “frontend ” essentially what the user interacts with)

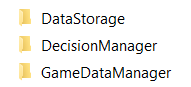
This is largely intertwined with the user interface as everything is split up into components. Maybe I can have a subheading for each major component. (see screenshot – not including component that haven’t been created yet.)



Logic

(what I refer to as a the “backend”, not referring to an actual backend in the sense of something that serves a frontend.)

This will be the bits that connect the components (listed above) and also the larger “systems” and “interfaces”



Fuzzy Logic Implementation

* C# and the Fuzzy Logic Sharp library.
* Problems encountered trying to implement fuzzy logic in JavaScript, then Ruby and then finally C#.

Testing

Remember to really hit home the testing and provide a good in-depth rationale of why you did what you did, how did this effect the final configuration of the system.

Automated Testing

* Process
* Tools Used

Tuning

Present the results of the testing in a meaningful way - the use of tables to highlight the important aspects of the testing. cherry pick what should be showcased in the main body of the report and place the rest in the appendices.

* Changes made as a result of testing and fine tuning the application

[Rough notes for tuning]

* Heavily biased political leaning of provinces to the centre of the spectrum. Initial attempt is to remove one of the centre-leaning rules :: Second attempt will be to adjust \_how\_ the random attributes are generated. – This was an issue with the randomly generated values, it tended to lean towards really high values and one of the centre rules is triggered if three out of the 4 attributes are VeryHigh, thus triggering this rule a lot.

User Testing

Remember to try and include user testing to demonstrate that you have listened to external input.

Documentation

Conclusion

* Around 7.5% of the overall report (introduction + conclusion should take up 15% of the report)
* Final Product Evaluation
  + How much does it do?
  + How good is it? (relative to what I set out to accomplish in my project contract?)
  + What hasn’t been implemented?
  + How would I extend the game if I had more time?
* Evaluation of approach
  + How was my experience approaching the project management using an agile methodology?
  + How was my experience approaching development using a Test Driven Development methodology (TDD)?
  + What did I learn by doing the project both academically and personally?
  + How would I approach the project if I started it again?
* Evaluation of tools used
  + Discussion around targeting the Web as a platform
  + Discussion around my use of React + TypeScript instead of other potential technologies
    - What went well
    - What could have been better
    - What didn’t I predict when choosing these technologies?
  + If I started again, would my choice of technology change?

Bibliography

12. deliberate practice: <https://www.calnewport.com/blog/2012/03/28/the-satisfying-strain-of-learning-hard-material-a-deliberate-practice-case-study/>

Figure 5. <https://dev.thenerdstash.com/hearts-of-iron-iv-review/>

Appendix